Wstęp do programowania

Jak pisać poprawne programy. Kilka ładnych rozwiązań.

Paweł Daniluk

Wydział Fizyki

Jesień 2013

Poważne decyzje

Problem

Przed przystąpieniem do pracy problem musi być dokładnie określony. Specyfikuje się rodzaj danych wejściowych, oczekiwany wynik oraz dodatkowe wymagania i ograniczenia.

Algorytm

Musi być dostosowany do wymagań.

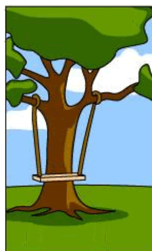
Struktury danych

Często reprezentacja danych jest determinowana przez wybrany algorytm, ale nie zawsze.

Bez określenia w/w nie należy przystępować do programowania.

Cykl rozwoju oprogramowania

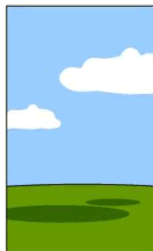
How the customer explained it

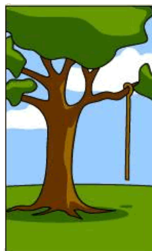

How the Project Leader understood it

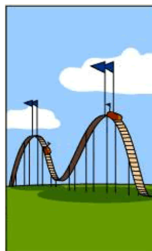
How the Analyst designed it

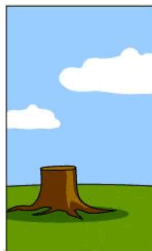
How the Programmer wrote it

How the Business Consultant described it


How the project was documented


What operations installed


How the customer was billed


How it was supported

What the customer really needed

Cykl rozwoju oprogramowania

1. Określenie wymagań
2. Projektowanie
3. Implementacja
4. Testy
5. Konserwacja

W praktyce wielokrotnie wraca się do faz poprzedzających i wprowadza korekty problemów ujawniających się podczas implementacji lub testów.

Przykra rzeczywistość

Praktycznie nie zdarza się, żeby program zadziałał poprawnie przy pierwszym uruchomieniu.

Przykra rzeczywistość

Praktycznie nie zdarza się, żeby program zadziałał poprawnie przy pierwszym uruchomieniu.

Jak pisać możliwie poprawne programy?
Jak szukać błędów?

Jak pisać możliwie poprawne programy?

Metoda małych kroków

1. Zacznij od małego programu, który robi coś widocznego.
2. Dodawaj po kilka linii kodu i za każdym razem sprawdzaj, jak działa.
3. Rób tak, aż skończysz.

Testy

Warto od razu obmyślać sposób testowania poszczególnych komponentów.

Każda zmiana jest przetestowana i poprawna.
Łatwo lokalizować błędy.

W dobrze zaprojektowanym programie komponenty są możliwie niezależne.

Modularyzacja

Jeżeli widzisz kawałek kodu, który robi coś spójnego i (być może) powtarzalnego, zrób z niego funkcję.

Wielokrotne występowanie podobnych fragmentów kodu powinno budzić zaniepokojenie.

Uogólnianie

1. dodanie parametrów
2. instrukcją warunkowa służąca do wyboru właściwej ścieżki
3. funkcje pomocnicze

Przykład – parametry

Przed

```
def inc_by_two(l):
    return [i+2 for i in l]

def inc_by_three(l):
    return [i+3 for i in l]
```

Po

```
def inc_by_number(l, n):
    return [i+n for i in l]

def inc_by_two(l):
    return inc_by_number(l, 2)

def inc_by_three(l):
    return inc_by_number(l, 3)
```

Przykład – instrukcja warunkowa

Przed

```
# fragment A
...
#

B1()

# fragment C
...
#

# fragment A
...
#

B2()

# fragment C
...
#
```

Po

```
def f(v):
        # fragment A
        ...
        #

        if v==1:
            B1()
        elif v==2:
            B2()

        # fragment C
        ...
        #

f(1)
f(2)
```

Przykład – funkcje pomocnicze

Przed

```
# fragment A
...
#

B1()

# fragment C
...
#

# fragment A
...
#

B2()

# fragment C
...
#
```

Po

```
def A():
    # fragment A
    ...
    #

def C():
    # fragment C
    ...
    #

A()
B1()
C()

A()
B2()
C()
```

Szybkie prototypowanie

W początkowej fazie można stosować rozwiązania uproszczone np. mniej wydajne.

Przykłady

- wyszukiwanie liniowe zamiast binarnego
- metoda "brute-force"
- pomijanie obsługi sytuacji nietypowych (np. błędów w danych wejściowych)

Częste błędy

Zbyt duże przyrosty

Duże kawałki nowego kodu są trudne do poprawiania.

Upór

Nie należy przywiązywać się do błędnego kodu.

Błądzenie losowe

Wprowadzanie losowych zmian donikąd nie prowadzi.

Uleganie komunikatom kompilatora/interpretera

Komunikaty o błędach mogą wprowadzać w błąd.

Projektowanie – dwie strategie

Top-down

Problem jest rozbijany na podproblemy, które można rozwiązać niezależnie. Podproblemy są dalej sukcesywnie dzielone do poziomu łatwo implementowalnych funkcji.

Bottom-up

Najpierw definiowane są podstawowe elementy, następnie składane są z nich coraz większe komponenty.

Projektowanie – dwie strategie c.d.

Top-down

- Brakujące elementy całości trzeba tymczasowo uzupełnić przed uruchomieniem.
- W naturalny sposób przechodzi od ogólnego zarysu do szczegółowych rozwiązań.
- Wymaga kompletnego projektu.
- Ułatwia delegowanie zadań i programowanie zespołowe.
- Pozwala poprawnie zaprojektować system od podstaw.

Bottom-up

- Szybko uzyskuje się fragmenty, które można testować.
- Trzeba z góry przewidzieć jakie komponenty są niezbędne.
- Można rozpocząć kodowanie bez ostatecznej koncepcji.
- Zwiększa prawdopodobieństwo, że bloki funkcjonalne będą uniwersalne.
- Pozwala rozwijać istniejący program.

Wypisywanie

Warto od razu przygotowywać funkcje służące do wypisywania danych.

Błędy składniowe

Typowe błędy

- SyntaxError: invalid syntax np.:
  - brak dwukropka po instrukcji `if`, albo `while`
  - próba użycia słowa kluczowego jako nazwy zmiennej
- IndentationError: expected an indented block
- IndentationError: unexpected indent
- NameError: global name '—' is not defined np:
  - użycie zmiennej bez inicjalizacji
  - literówka (np. "imput" zamiast "input"
- TypeError: Can't convert 'int' object to str implicitly

Błędy wykonania

Śledzenie przebiegu wykonania

`print` doskonale się do tego nadaje.

Program się zawiesza

Nieskończona pętla lub rekurencja.
Należy zlokalizować problematyczną pętlę i sprawdzić co się dzieje z warunkiem.

```
while x > 0 and y < 0: \{
    // do something to x
    // do something to y

    print "x:", x, "y:", y
    print "condition:",(x > 0 and y < 0)
```

Błędy wykonania – wyjątki

Błędy arytmetyczne

- **OverflowError** przepełnienie – obliczona wartość jest zbyt duża
- **ZeroDivisionError** dzielenie przez zero
- **FloatingPointError** błąd operacji zmiennoprzecinkowej

Brak elementu

- **IndexError** w liście
- **KeyError** w słowniku

- **NameError** zmienna/funkcja nie istnieje
- **AttributeError** atrybut nie istnieje

- **TypeError** błąd typu (np. liczba w zamiast napisu)

- **KeyboardInterrupt** naciśnięto Ctrl+C

Błędy wykonania – kiedy wszystko zawodzi

Zmniejsz rozmiar danych.

Usuń nieistotne fragmenty kodu. Znajdź minimalną wersję, która zawiera błąd.

Jeżeli zmiana, która nie powinna mieć efektu, zmienia działanie programu, należy zwrócić na to uwagę.

Błędy logiczne – program działa, ale źle

Należy zrozumieć, co program naprawdę robi.

- Czy program ewidentnie powinien coś robić, ale nie robi?
- Czy robi coś czego nie powinien?
- Czy jakiś kawałek kodu daje niespodziewany rezultat?

Częste błędy

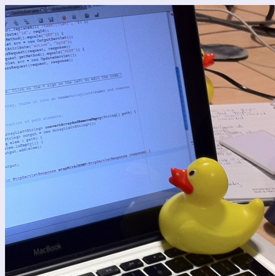
- Dzielenie liczb całkowitych daje całkowity wynik.
- Operacje na liczbach zmiennoprzecinkowych nie są dokładne.
- Operator przypisania (=) zamiast (==).

Kiedy wszystko zawodzi

Komputery emitują fale zaburzające pracę mózgu, które powodują:

- frustrację i gniew
- przesądność ("komputer mnie nienawidzi") i myślenie magiczne ("program działa wyłącznie, gdy trzymam się za lewe ucho")
- pesymizm ("ten program jest beznadziejny")

Metoda gumowej kaczki



Stacje benzynowe

Problem

Na zamkniętym torze znajduje się N stacji benzynowych. W ich zbiornikach jest w sumie dokładnie tyle paliwa, ile potrzeba do przejechania całego toru. Należy wskazać stację (o ile istnieje), od której można zacząć jazdę z pustym bakiem tak, aby tankując całe paliwo na odwiedzanych stacjach można było przejechać dookoła toru. Zakładamy, że samochód ma nieograniczenie duży zbiornik.

Dane wejściowe

- liczba stacji
- odległości pomiędzy stacjami (położenie stacji na torze)
- ilość paliwa na stacjach

Stacje benzynowe c.d.

Metoda

Jeżeli istnieje stacja i, z której startując z pustym bakiem można dojechać do następnej stacji $(i + 1)$, to rozwiązania zadania dla układu, w którym paliwo z $i + 1$ jest w i, jest rozwiązaniem dla problemu wyjściowego.

Stacje benzynowe c.d.

Metoda

Jeżeli istnieje stacja i, z której startując z pustym bakiem można dojechać do następnej stacji $(i + 1)$, to rozwiązania zadania dla układu, w którym paliwo z $i + 1$ jest w i, jest rozwiązaniem dla problemu wyjściowego.

Algorytm

1. Znaleźć odpowiednią parę kolejnych stacji
2. Przelać paliwo
3. Tak postępować, aż całe paliwo będzie na jednej stacji.

Stacje benzynowe c.d.

Metoda

Jeżeli istnieje stacja i, z której startując z pustym bakiem można dojechać do następnej stacji $(i+1)$, to rozwiązania zadania dla układu, w którym paliwo z $i+1$ jest w i, jest rozwiązaniem dla problemu wyjściowego.

Algorytm

1. Znaleźć odpowiednią parę kolejnych stacji
2. Przelać paliwo
3. Tak postępować, aż całe paliwo będzie na jednej stacji.

Nie wiadomo, które stacje wybierać, ani jak je znajdować.

Stacje benzynowe c.d.

Reprezencacja danych

n	liczba stacji
dist	lista zawierająca odległości
fuel	lista zawierająca ilość paliwa

Rozwiązanie chaotyczne

```
n=5
dist=[23,45,13,33,30]
fuel=[30,23,45,13,33]

cnt=n

while cnt>1:
    for i in range(n):
        if fuel[i]>0:
            d=dist[i]
            pos=(i+1)%n   #next(i)
            while fuel[pos]==0: # next_nonempty(i)
                d+=dist[pos]
                pos=(pos+1)%n   #next(i)

            if fuel[i]>=d:
                fuel[i]+=fuel[pos]
                fuel[pos]=0
                cnt-=1
print dist
print fuel
```

DRY - Don't Repeat Yourself

```
def next(i):
    return (i+1)%n


while cnt>1:
    for i in range(n):
        if fuel[i]>0: # fix_station(i)
          d=dist[i]
          pos=next(i)
          while fuel[pos]==0: # next_nonempty(i)
              d+=dist[pos]
              pos=next(pos)

          if fuel[i]>=d:
              fuel[i]+=fuel[pos]
              fuel[pos]=0
              cnt-=1
```

Modularyzacja

```
def next(i):
    return (i+1)%n

def next_nonempty(i):
    d=dist[i]
    pos=next(i)
    while fuel[pos]==0:
        d+=dist[pos]
        pos=next(pos)
    return (pos, d)

def pump(to, fr):
    fuel[to]+=fuel[fr]
    fuel[fr]=0

while cnt>1:
    for i in range(n):
        if fuel[i]>0:
            pos, d = next_nonempty(i)

            if fuel[i]>=d:
                pump(i, pos)
                cnt-=1
```

Optymalizacja 1

```
def next_unreachable(i):
    d=dist[i]
    f=fuel[i]
    pos=next(i)
    while f>=d and pos!=i:
        d+=dist[pos]
        f+=fuel[pos]
        pos=next(pos)
    return pos

while cnt>1:
    for i in range(n):
        if fuel[i]>0:
            pos = next_unreachable(i)

            j=next(i)
            while j!=pos:
                pump(i, j)
                cnt-=1
                j=next(j)
```

Zamiast szukać niepustej stacji można szukać pierwszej, do której nie można dojechać.

Optymalizacja 2

```
while cnt>1:
    pos = next_unreachable(i)

    print i, pos

    j=next(i)
    while j!=pos:
        pump(i, j)
        cnt-=1
        j=next(j)
```

Nie trzeba sprawdzać stacji już opróżnionych.

Optymalizacja 3

```
def next_unreachable(i):
    d=dist[i]
    f=fuel[i]
    pos=next(i)
    cnt=0
    while f>=d and pos!=i:
        d+=dist[pos]
        f+=fuel[pos]
        pump(i, pos)
        cnt+=1
        pos=next(pos)
    return pos, cnt

i=0
while cnt>1:
    pos, c = next_unreachable(i)
    cnt-=c
    i=pos
```

Można przepompowywać paliwo w momencie sprawdzania, czy da się pojechać dalej.

Optymalizacja 4

```
nsta=range(1,5)+[0]

def next(i): return nsta[i]

def next_unreachable(i):
    pos=next(i)
    cnt=0
    while fuel[i]>=dist[i] and pos!=i:
        pump(i, pos)
        cnt+=1
        pos=next(pos)
    return pos, cnt

def pump(to, fr):
    dist[to]+=dist[fr]
    fuel[to]+=fuel[fr]
    nsta[to]=nsta[fr]
    fuel[fr]=0

i=0
while cnt>1:
    pos, c = next_unreachable(i)
    cnt-=c
    i=pos
```

Ostateczny program

```
nsta=range(1,5)+[0]

cnt=n

def next(i): return nsta[i]

def pump(to, fr):
    dist[to]+=dist[fr]
    fuel[to]+=fuel[fr]
    nsta[to]=nsta[fr]
    fuel[fr]=0

i=0
while cnt>1:
    pos=next(i)
    if fuel[i]>=dist[i]:
        pump(i, pos)
        cnt-=1
        pos=next(pos)

    i=pos
```

Strona wykładu

http://bioexploratorium.pl/wiki/Wstęp_do_programowania_
_2013z